1988

Automated Prototyping Data and Knowledge Translation

Luqi

Naval Postgraduate School

Luqi, "Automated Prototyping Data and Knowledge Translation", Technical Report NPS 52-88-034, Computer Science Department, Naval Postgraduate School, 1988.
http://hdl.handle.net/10945/65286

NPS52-88-034

NAVAL POSTGRADUATE SCHOOL

Monterey, California

AUTOMATED PROTOTYPING DATA AND KNOWLEDGE TRANSLATION

Luqi

September 1988

Prepared for:

Naval Postgraduate School
Monterey, CA. 93943

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. C. Austin
Superintendent

H. Shull
Provost

The work reported herein was supported in part by the National Science Foundation and the Naval Postgraduate School Research Foundation.

This report was prepared by:

LUQI
Assistant Professor
of Computer Science

Reviewed by:

ROBERT B. MCGHEE
Chairman
Department of Computer Science

Released by:

KNEALE T. MARSHALL
Dean of Information
and Policy Science

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPS52-88-034	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL (*If applicable*) 52	7a. NAME OF MONITORING ORGANIZATION National Science Foundation
6c. ADDRESS (*City, State, and ZIP Code*) Monterey, CA. 93943	7b. ADDRESS (*City, State, and ZIP Code*) Washington, DC 20550
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Naval Postgraduate School 8b. OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER O&MN, Direct funding
8c. ADDRESS (*City, State, and ZIP Code*) Monterey, CA. 93943	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (*Include Security Classification*)

AUTOMATED PROTOTYPING DATA AND KNOWLEDGE TRANSLATION (U)

12. PERSONAL AUTHOR(S)
LUQI

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
Progress	FROM 87/10 TO 88/09	1988, Sept	20

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	rapid prototyping, attribute grammars, translator generator, Ada^2, specification language, real-time systems, embedded system design, application generator, computer aided soft-

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

This paper describes the transformations used to produce an executable prototype from a very high level description of a software system in a computer-aided prototyping process. The high level description can include real-time constraints, which are difficult to treat using conventional compiler technology. The prototyping system uses two different sets of transformations, one for realizing data flow diagrams and the other for realizing hard real-time constraints. The transformations are implemented with the aid of an automatic translator generator.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Luqi	22b. TELEPHONE (*Include Area Code*) (408) 646-2735 22c. OFFICE SYMBOL 52Lq

18. ware engineering

Automated Prototyping Data and Knowledge Translation[1]

Luqi

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

ABSTRACT

This paper describes the transformations used to produce an executable prototype from a very high level description of a software system in a computer-aided prototyping process. The high level description can include real-time constraints, which are difficult to treat using conventional compiler technology. The prototyping system uses two different sets of transformations, one for realizing data flow diagrams and the other for realizing hard real-time constraints. The transformations are implemented with the aid of an automatic translator generator.

KEYWORDS

Rapid prototyping, attribute grammars, translator generator, Ada[2], specification language, real-time systems, embedded system design, application generator, computer aided software engineering

1. Introduction

We believe computer aided prototyping shows promise. Prototyping is an effective way to establish accurate requirements because customers can usually recognize what will or won't solve their problems when they see it demonstrated, even though they often cannot clearly describe the system they want. The use of prototyping in requirements analysis works best if prototypes can be constructed and modified rapidly. The key to speeding up the prototyping process is reducing the amount of work that must be done by the prototype designers via abstraction and automation. Our approach to rapid prototyping uses a high level prototyping language (PSDL - Prototype System Description Language) [14] integrated with a prototyping method [11] and a set of software tools (CAPS - Computer Aided Prototyping System) [12] to achieve this goal. This approach frees designers from many implementation details by constructing executable specifications from reusable components maintained by a software design management system. PSDL and CAPS are particularly intended for supporting the rapid prototyping of complex real-time and embedded systems because the requirements of such systems are difficult to develop and understand without

[1]This research was supported in part by the National Science Foundation under grant number CCR-8710737.

[2]Ada is a registered trademark of the United States Government Ada Joint Program Office.

access to executable models. PSDL and CAPS include special facilities for describing real-time systems.

Data and knowledge translation plays an important part in our approach because the designer's view of the prototype is expressed at a high level that is significantly simpler than a programming language level description of the prototype. The prototyping language PSDL is the central representation for specification and design aspects of the prototype, which is used both for interacting with the designer and with the design management system. The notation was designed primarily to support simple abstract descriptions that can be readily understood by the designer and support the retrieval of reusable software components with minimal designer effort. These goals dictate a structure for PSDL that is much closer to the problem domain than to the underlying machine. The data and knowledge expressed in a PSDL design must be transformed into detailed algorithmic form to allow the execution of the prototype. This makes translation an important part of the processing performed by the CAPS system.

The tools in CAPS include user interfaces to speed up design entry and prevent syntax errors, an execution support system to demonstrate and measure prototype behavior and to perform static analyses of the prototype design, a software design-management system to manage reusable software components and design data, a software base to store reusable components, and a design database to store the prototype design. The prototyping language is an integral part of the CAPS software tools, since many of the CAPS tools operate on PSDL. For example, PSDL specifications are used by the software design management system for organizing and retrieving Ada reusable components in the software base. Translations are needed to adapt the information in the PSDL descriptions to the needs of individual CAPS tools.

PSDL is optimized for use at the specification and design level. The prototyping language allows the designer to use dataflow diagrams with non-procedural control constraints as part of the specification of a hierarchically structured prototype. The resulting description is free from programming level details, in contrast to prototypes constructed using a programming language. The structure of the language encourages modular design of the prototype, and by extension, of the eventual production version. The underlying computational model unifies data flow and control flow, providing a vehicle suitable for developing top-down decompositions. Such decompositions allow large prototypes to be executed with practical computation times, in contrast to prototyping by simulating specifications via logic programming without providing a system architecture [13].

Abstractions are an important means for controlling complexity [2]. This is especially important in rapid prototyping because a system must appear to be simple to be built or analyzed quickly. PSDL supports three kinds of abstractions: **operator abstractions, data abstractions,** and **control abstractions.** The first two kinds of abstractions correspond to the fundamental building blocks for PSDL prototype descriptions: operators and data types. A PSDL description represents a system decomposition as operators communicating via data streams. Each data stream carries values of a fixed abstract data type. Each data stream can also contain values of the built-in type "exception". The operators may be either data driven or periodic. Periodic operators have traditionally been the basis for most real-time system design, while the importance of data driven operators for real-time systems is recognized [9]. Such a description is based on the PSDL computational model. Formally the computational model is an augmented graph

$$G = (V, E, T(v), C(v))$$

where V is the set of vertices, E is the set of edges, T(v) is the maximum execution time for each vertex v, and C(v) is the set of control constraints for each vertex v. Each vertex is an **operator** and each edge is a **data stream.** The first three components of the graph are called the enhanced data flow diagram.

Control abstractions correspond to the non-procedural control constraints of PSDL. Control constraints are combined with an enhanced data flow diagram to define the implementation of a composite operator. Conditional execution is supported by PSDL triggering conditions and conditional outputs.

Prototyping languages support executable specifications. There are two approaches to making a prototyping language executable, one based on meta-programming and the other on executable specifications [1]. The PSDL prototyping language uses the first approach -- PSDL programs contain specifications of the desired systems' behavior and an optional description of the system's structure. A PSDL prototype description is an executable program whose behavior can be tested if all required information is supplied. This information is distributed between the program and a software base containing reusable software components. The software base contains implementations for all atomic operators and types. The language uses the enhanced dataflow diagram as a basis for combining operators. Control constraints and timing constraints are the fundamental mechanisms to aid execution. Ada is used for implementing both the PSDL reusable components in the software base and the PSDL execution support environment.

2. Data and Knowledge Transformations in CAPS

The implementation of PSDL relies on several kinds of data and knowledge transformations. The PSDL execution support system contains a translator, static scheduler, and a dynamic scheduler [10]. The translator generates code binding together reusable components extracted from the software base. Its main functions are to implement data streams and control constraints. The static scheduler allocates time slots for operators with real-time constraints. If the allocation succeeds, all operators are guaranteed to meet their deadlines even with worst-case execution times. The dynamic scheduler invokes operators without real-time constraints in the time slots not used by the operators with real-time constraints [3]. The execution support system frees the designer from the implementation effort required in Ada by automatically generating executable code in Ada, and by automatically generating control code in the form of static and dynamic schedules which enforce control and timing behavior. The complete Ada source program corresponding to a PSDL prototype consists of the code produced by the translator and static scheduler together with the definitions of the reusable components retrieved from the software base and some fixed run-time support code.

The main data and knowledge transformations in the PSDL execution support system are embodied in the translator and static scheduler. The translator transforms PSDL augmented data flow diagrams and the associated control constraints into the corresponding Ada code. The output of the translator is just part of the implementation of a PSDL prototype. PSDL operators can be either atomic or composite. During the early prototype design phase, PSDL composite operators are decomposed into atomic operator networks. Atomic operators are realized by reusable modules from the CAPS software database, in the form of Ada program units. The code generated by the translator serves to adapt and connect the atomic operators realizing each composite operator.

The static scheduler transforms PSDL timing constraints into a static schedule. The static schedule is a piece of Ada code containing calls to the program units implementing the operators to be scheduled. The static schedule acts as an executive that invokes operators with hard real-time constraints as needed to enforce timing constraints at run-time. The static schedule takes into account the worst case time requirements for all operators that have real-time constraints such as a maximum execution time, minimum calling period, and minimum response time.

2.1. Correspondence Between PSDL Operator Networks and Ada

This section describes the correspondence between PSDL and Ada used by the translator [15]. This mapping gives the syntactic and semantic correspondence between the two languages.

2.1.1. PSDL Operators

An **operator** is either a function or a state machine. When an operator fires, it reads one data object from each of its input streams, and writes at most one data object on each of its output streams. The output objects produced when a function fires depend only on the current set of input values. The output values produced when a state machine fires depend on the current set of input values and the current values of a finite number of internal state variables.

In the simplest case PSDL operators are implemented by Ada procedures. These procedures contain code to implement input and output to PSDL data streams, PSDL triggering conditions, and PSDL conditional output statements. This includes both functions and state machines. The state variables of composite state machines are realized by PSDL data streams connected to form feedback loops. Atomic state machines are reusable components, which are usually realized as procedures embedded in Ada packages, where the declaration of the procedure is public and the declarations for the state variables are private.

2.1.2. PSDL Data Streams

A **data stream** is a communication link connecting one or more producer operators to a consumer operator. Each stream carries a sequence of data values. Communication links with more than two ends are realized using implicit copy operators.

There are two types of data streams - **dataflow** streams and **sampled** streams. A dataflow stream guarantees that none of the data values are lost or replicated, while a sampled stream guarantees the most recently generated data value is always available. Dataflow streams are used to connect operators that must coordinate corresponding input values from different producers. Sampled streams are used to connect operators that fire at incompatible frequencies.

A PSDL stream is mapped into a buffer capable of holding one data value. Since a buffer may be read by an operator executing independently of the operator writing into the buffer, it must be protected

from data conflicts due to concurrent access. Consequently buffers are embedded in Ada tasks and read or written via task entries, to provide mutually exclusive access. Buffer manager tasks are declared inside generic packages to make it easy for the translator to create a separate buffer manager task for each PSDL data stream. Thus each PSDL data stream is implemented by an instance of a generic package.

Two kinds of buffers are needed, corresponding to the two kinds of data streams in PSDL. Sampled buffers are used to implement sampled streams and FIFO buffers are used to implement dataflow streams. The difference between the two kinds of buffers is that a FIFO buffer makes sure that every value written into the buffer is read exactly once before the next value is written into the buffer. Violations of this constraint are reported via Ada exception conditions. There are two possible exceptions: Underflow and Overflow. Underflow is raised if the consumer operator attempts to read the buffer before it has been updated by the producer operator. Overflow is raised if the producer attempts to write to the buffer before the consumer has read the previous data value. There are no constraints on the order a sampled buffer is accessed, and no associated exception conditions.

The translator must select the appropriate type for buffer for a given data stream according to the triggering conditions of the consumer operator associated with the stream. There are two types of **data triggers** for PSDL operators.

```
OPERATOR p TRIGGERED BY ALL x, y, z
OPERATOR q TRIGGERED BY SOME a, b
```

In the first example the operator p is ready to fire whenever new data values have arrived on all three of the input arcs x, y, and z. In the second example, the operator q fires when any of the inputs a and b gets a new value. If q has some other input c, the output of q can be based on old values of c, since q will not be triggered on a new value of c until after a new value for a or b arrives.

If an operator mentions an input data stream in a TRIGGERED BY ALL condition then the translator will use a FIFO buffer to realize the stream, and otherwise it will use a sampled buffer. The translator realizes each sampled buffer as an instance of the generic package "sampled_buffer" and each FIFO buffer as an instance of the generic package "fifo_buffer". These generic packages are standard reusable components contained in the software base for PSDL. Simplified versions of these modules are shown in Fig. 1

and Fig. 2. The full implementations of these modules include additional operations for writing exception values into the buffer and checking whether the current value in the buffer represents an exception.

The translator also generates the definition of a procedure for initializing the buffers realizing data streams whose initial values have been declared in PSDL. This is done by using the write operation provided by each buffer task. The procedure body contains one such statement for each data stream with a declared initial value, and can be empty if no initial values are declared. Typically initial values are declared for streams implementing the state variables of composite machines.

```
generic type ELEMENT_TYPE is private;
package SAMPLED is
 task SAMPLED_BUFFER is
  entry CHECK (NEW_DATA : out BOOLEAN);
  entry PUT (VALUE : in ELEMENT_TYPE);
  entry GET (VALUE : out ELEMENT_TYPE);
 end SAMPLED_BUFFER;
end SAMPLED;

package body SAMPLED is
 task body SAMPLED_BUFFER is
  BUFFER : ELEMENT_TYPE;
  VALUE : ELEMENT_TYPE;
  NEW_DATA_VALUE: BOOLEAN := false;
 begin
  loop
   select
    accept CHECK (NEW_DATA : out BOOLEAN) do
     NEW_DATA := NEW_DATA_VALUE;
    end CHECK;
   or
    accept GET (VALUE : out ELEMENT_TYPE) do
     VALUE := BUFFER; NEW_DATA_VALUE := false;
    end GET;
   or
    accept PUT (VALUE : in ELEMENT_TYPE) do
     BUFFER := VALUE; NEW_DATA_VALUE := true;
    end PUT;
   end select;
  end loop;
 end SAMPLED_BUFFER;
end SAMPLED;
```

Fig. 1 A Sampled Stream Buffer Task

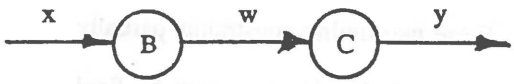
```
generic type ELEMENT_TYPE is private;
package FIFO is
 task FIFO_BUFFER is
  entry CHECK (NEW_DATA : out BOOLEAN);
  entry PUT (VALUE : in ELEMENT_TYPE);
  entry GET (VALUE : out ELEMENT_TYPE);
 end FIFO_BUFFER;
 BUFFER_READ_ERROR, BUFFER_WRITE_ERROR : exception;
end FIFO;

package body FIFO is
 task body FIFO_BUFFER is
  BUFFER : ELEMENT_TYPE;
  VALUE : ELEMENT_TYPE;
  NEW_DATA_VALUE: BOOLEAN := false;
 begin
  loop
   select
    accept CHECK (NEW_DATA : out BOOLEAN) do
     NEW_DATA := NEW_DATA_VALUE;
    end CHECK;
   or
    accept GET (VALUE : out ELEMENT_TYPE) do
     if NEW_DATA_VALUE then
       VALUE := BUFFER; NEW_DATA_VALUE := false;
     else raise BUFFER_READ_ERROR; end if;
    end GET;
   or
    accept PUT (VALUE : in ELEMENT_TYPE) do
     if not NEW_DATA_VALUE then
       BUFFER := VALUE; NEW_DATA_VALUE := true;
     else raise BUFFER_WRITE_ERROR; end if;
    end PUT;
   end select;
  end loop;
 end FIFO_BUFFER;
end FIFO;
```

Fig. 2 A FIFO Buffer Task

2.1.3. An Example of the Mapping between PSDL and Ada

This section gives an example of the transformation performed by the translator. Consider the PSDL description of the composite operator A shown in Fig. 3. The Ada code generated by the translator for the operator B used in the implementation of A is shown in Fig. 4. This example assumes the procedure b is a reusable component implementing the PSDL operator B. The procedure b_driver contains the augmentation code generated by the translator and is called from the "static_schedule" every 100 milliseconds.

```
OPERATOR A
 SPECIFICATION
  INPUT x: integer
  OUTPUT y: integer
 IMPLEMENTATION
  GRAPH
```




```
   OPERATOR B
    TRIGGERED IF x > 0
    PERIOD 100 ms
END
```

Fig. 3 Sample PSDL Operator

```
x_buffer is new sampled_buffer(integer);  -- instance of generic package

procedure b_driver is
 x, w: integer;
begin
 if x_buffer.new_data then  -- new data in the buffer
   x_buffer.read(x);
   if x > 0 then  -- triggering condition is true
         b(x, w);
         w_buffer.write(w);
   end if;
 end if;
end b_driver;
```

Fig. 4 Sample Ada Implementation of a PSDL Operator

2.2. Correspondence between PSDL Timing Constraints and Ada

Any PSDL operator can have timing constraints associated with it. An operator is **time-critical** if it has at least one timing constraint associated with it, and is **non time-critical** otherwise. The static scheduler is concerned only with the time-critical operators in a PSDL prototype. All PSDL timing constraints can be represented by constants denoting lengths of time intervals. There are several different kinds of timing constraints, which can be classified into those that apply to all time-critical operators, those that apply only to operators triggered by periodic temporal events, and those that apply only to operators

triggered by the arrival of new data. Temporal events occur at specified absolute times.

Every time-critical operator must have a **maximum execution time** (MET) to allow the construction of a static schedule. The MET of an operator is an upper bound on the length of the **execution interval** for the operator. All of the actions that may be required to fire an operator once must fit into the execution interval.

Operators triggered by temporal events are periodic in PSDL. Every periodic operator must have a **period** (PERIOD) and may have a **deadline** (FINISH_WITHIN). These two timing constraints partially determine the set of **scheduling intervals** (SI) for the operator. Each periodic operator must be fired exactly once in each scheduling interval, and must complete execution before the end of the scheduling interval. The period is the length of time between the start of any scheduling interval and the start of the next scheduling interval. The deadline is the length of each scheduling interval.

Operators triggered by the arrival of new data values are sporadic. Timing constraints for sporadic operators are optional. Sporadic operators with timing constraints must have both a **maximum response time** (MRT) and a **minimum calling period** (MCP) in addition to an MET. The MRT is an upper bound on the **response time**, while the MCP is a lower bound on the **calling period**. The response time associated with a consumer operator is measured from the end of the execution interval for the producer operator of the triggering data value to the end of the execution interval for the consumer operator of the triggering data value. The calling period of an operator is the length of time between the end of the execution interval for the producer of the triggering data value and the end of the execution interval for the producer of the next triggering data value.

The static schedule for the time-critical operators in a program gives the execution intervals for each time-critical operator. The static schedule contains a timing pattern of finite length which is repeated indefinitely. Sporadic operators are implemented by their periodic equivalents to fit into this framework.

The static scheduler consists of a data transformation and a knowledge transformation. The data transformation extracts the names of the time-critical operators, the associated timing constraints, and the data flow graphs for these operators from the PSDL source. This data transformation is defined using attribute equations.

The knowledge transformation is embodied in a heuristic algorithm for deriving a schedule from the timing constraints and the flow constraints [7, 16] which produces a valid schedule whenever it terminates without reporting an error. The flow constraints are used to determine a scheduling order for the initial firings of each operator which ensures any operator producing a data value is scheduled before the operators that consume the data value.

3. Implementation of the Data Transformations

The data transformations are realized using an automated translator generator called Kodiyak which was developed at the University of Minnesota [4, 5]. It is available as a research tool and is quite effective. The system is based on Knuth's attribute grammars [8]. It utilizes a variation of Jalili's algorithm [6] to evaluate the semantic tree it creates when generating a translation. More details about generating translators using attribute grammars can be found in [17].

The input to the Kodiyak tool is an attribute grammar describing the desired translation. This translator description defines the terminal and non-terminal tokens of the source language, declares the types of the attributes of each token, lists the productions of a context free grammar for the source language, and gives a set of attribute equations for each production. The attribute equations describe the relationship between the source language (in this case PSDL) and the target language (in this case Ada). The Kodiyak translator generator system utilizes these equations to produce C, Yacc, and Lex specifications that are compiled to produce an executable translator. The resulting translator program takes an input file in the source language (PSDL) and produces an output file containing the derived code in the target language (Ada). The process of generating a translator using Kodiyak is illustrated in Fig. 5.

The attribute equations defining the translator map PSDL constructs to the constructs of the target language Ada according to the mappings described in the previous section. Fig. 6 illustrates some of the attribute equations used by the Kodiyak translator generator to produce the PSDL to Ada translator. The attribute equations associated with each production are enclosed in set braces "{ }". The attribute "trn" represents the translation of a PSDL expression, while the attribute "%text" is the concrete text string in the input file corresponding to a terminal symbol of the source language such as "NUMBER". In Kodiyak "[x, y]" represents the concatenation of the strings x and y. The fragment of the attribute grammar shown

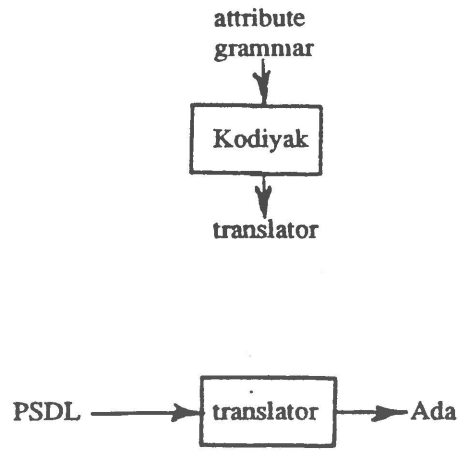


Fig. 5 Generating a Translator using Kodiyak

```
time
    : NUMBER unit
      {time.trn = [NUMBER.%text, unit.trn]; }
    ;

unit
    : MICROSEC
      {unit.trn = ""; }
    | MS
      {unit.trn = "000"; }
    ;
```

Fig. 6 Sample Attribute Equations for the PSDL Translator

defines the reduction of time expressions in arbitrary time units to a uniform representation where all time intervals are measured in microseconds. Only a small subset of the time units supported by PSDL are shown.

4. Conclusion and Future Research

Currently the CAPS system is under development as a set of separate components. Conceptual work has been completed for the design of both the static and dynamic schedulers. The feasibility of the translator and static scheduler has been demonstrated empirically via an initial implementation, and work is underway to improve the initial version and integrate it with other components of CAPS.

The present version of Kodiyak used to generate the translator is an excellent tool. It generates an effective easily modified translator. However, some improvement in the error messages returned to the user when the translator is applied to a syntactically incorrect input file is needed. The current versions of the translators do not do any semantic error checking. As the system develops, syntactic error recovery rules and additional attributes and attribute equations for semantic checks will have to be added to improve the robustness of the PSDL translator and prototypes.

Efforts to integrate the various parts of CAPS are waiting for development to proceed on remaining portions of the system. At present work has commenced on the Software Base management System at the conceptual and, to a limited degree, the empirical levels. Work is underway to develop the syntax directed editor for the system and portions of the graphic interface. As the remaining portions of the system are developed work will be required to combine all the individual tools into an integrated work environment.

Automated facilities to translate a prototyping language into an underlying implementation language are feasible, and are a working reality at this time. Much work remains to develop a completely integrated version of the computer-aided prototyping system. The aims of this research effort are to create a sound foundation for the development and use of highly automated program development environments. Such environments are designed to improve productivity in software development and are a first small step on the way to fully automatic generation of complete programs from specifications. Significant advances in theory and practice are needed before such a goal can be realized.

1. V. Berzins and Luqi, "Languages for Specification, Design and Prototyping", in *Handbook of Computer-Aided Software Engineering*, Van Nostrand Reinhold, 1988.

2. V. Berzins and Luqi, *Software Engineering with Abstractions: An Integrated Approach to Software Development using Ada*, Addison-Wesley, 1988.

3. S. Eaton, "An Implementation Design of A Dynamic Scheduler for a Computer Aided Prototyping System", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, March 1988.

4. R. Herndon, "The Incomplete AG User's Guide and Reference Manual", Tech. Rep. 85-37, University of Minnesota, October, 1985.

5. R. Herndon and V. Berzins, "The Realizable Benefits of a Language Prototyping Language", *IEEE Trans. on Software Eng. SE-14*, 6 (June 1988).

6. F. Jalili, "A General Linear-Time Evaluator for Attribute Grammars", *ACM SIGPLAN Notices Notices 18*, 9 (September 1983), 35-44.

7. D. Janson, "A static Scheduler for Hard Real-Time Constraints in the Computer Aided Prototyping System", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, March 1988.

8. D. E. Knuth, "Semantics of Context-free Language", *Math. Syst. Theory* 2, 2 (June 1968), 127-145.

9. Luqi, "Rapid Prototyping for Large Software System Design", Ph. D. Thesis, University of Minnesota, 1986.

10. Luqi, "Execution of Real-Time Prototypes", in *ACM First International Workshop on Computer-Aided Software Engineering*, vol. 2 , ACM, Cambridge, Massachusetts, May 1987, 870-884. Technical Report NPS 52-87-012.

11. Luqi and V. Berzins, "Rapid Prototyping of Real-Time Systems", *IEEE Software*, Sep. 1988.

12. Luqi and M. Ketabchi, "A Computer Aided Prototyping System", *IEEE Software 5*, 2 (March 1988), 66-72.

13. Luqi, "Specificaton Languages in Computer Aided Software Engineering", IEEE Software Design and Network Conference, Santa Clara, CA, April 1988.

14. Luqi, V. Berzins and R. Yeh, "A Prototyping Language for Real-Time Software", *IEEE Trans. on Software Eng.*, October, 1988.

15. C. Moffitt, "Development of a Language Translator for a Computer Aided Prototyping System", M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, March 1988.

16. J. O'Hern, ''A Conceptual Design of a Static Scheduler for Hard Real-Time Systems'', M. S. Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, March 1988.

17. T. Reps, ''Generating Language Based Environments'', Ph. D. Thesis, University of Massachusetts, Amherst, 1983.

Initial Distribution List

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943	2
Center for Naval Analysis 4401 Ford Avenue Alexandria, VA 22302-0268	1
Office of the Chief of Naval Operations Code OP-941 Washington, D.C. 20350	2
Office of the Chief of Naval Operations Code OP-945 Washington, D.C. 20340	2
Commander Naval Telecommunications Command Naval Telecommunications Command Headquarters 4401 Massachusetts Avenue NW Washington, D.C. 20390-5290	2
Commander Naval Data Automation Command Washington Navy Yard Washington, D.C. 20374-1662	1
Office of Naval Research Office of the Chief of Naval Research Attn. CDR Michael Gehl, Code 1224 Arlington, VA 22217-5000	1
Director, Naval Telecommunications System Integration Center NAVCOMMUNIT Washington Washington, D.C. 20363-5100	1
Space and Naval Warfare Systems Command Attn: Dr. Knudsen, Code PD50 Washington, D.C. 20363-5100	1

Ada Joint Program Office OUSDRE(R&AT) The Pentagon Washington, D.C. 230301	1
Naval Sea Systems Command Attn: CAPT Joel Crandall National Center #2, Suite 7N06 Washington, D.C. 22202	1
Office of the Secretary of Defense Attn: CDR Barber The Star Program Washington, D.C. 20301	1
Naval Ocean Systems Center Attn: Linwood Sutton, Code 423 San Diego, CA 92152-5000	1
National Science Foundation Division of Computer and Computation Research Washington, D.C. 20550	1
Director of Research Administration Code 012 Naval Postgraduate School Monterey, CA 93943	1
Chairman, Code 52 Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	1
LuQi Code 52Lq Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	150